Scoping Planning Agents With Shared Models

Tania Bedrax-Weiss* Jeremy D. Frank
Ari K. Jónsson† Conor McGann*
NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035-1000,
{tania,frank,jonsson,cmcgann}@email.arc.nasa.gov

Abstract

In this paper we provide a formal framework to define the scope of planning agents based on a single declarative model. Having multiple agents sharing a single model provides numerous advantages that lead to reduced development costs and increase reliability of the system. We formally define planning in terms of extensions of an initial partial plan, and a set of flaws that make the plan unacceptable. A Flaw Filter (FF) allows us to identify those flaws relevant to an agent. Flaw filters motivate the Plan Identification Function (PIF), which specifies when an agent is is ready hand control to another agent for further work. PIFs define a set of plan extensions that can be generated from a model and a plan request. FFs and PIFs can be used to define the scope of agents without changing the model. We describe an implementation of PIFsand FFswithin the context of EUROPA, a constraint-based planning architecture, and show how it can be used to easily design many different agents.

ToDo list

- spellcheck
- pagecount (8 pages)
- clean up latter parts of theory section
- related work needs to be reviewed since no longer execution focus
- discussion and future work

Introduction

Multi-agent systems are usually composed of multiple agents with different capabilities that cooperate to solve a problem. The problem decomposition, and therefore the agent architecture, can be motivated by different desires. A common example of such a system is a planner that collaborates with an executive system. This architecture is common due to uncertainty in the execution environment; the planner postpones decisions until execution time to avoid making poor predictions that will have to be undone. Another example is a science planner that determines the activities of a science instrument, and a tactical planner that integrates the science plan with a plan for other parts of a spacecraft, such as the power system and the attitude control system. This architecture may be motivated by a desire to plan for critical science goals and let these drive the rest of the planning.

When designing a multi-agent system, engineers typically create separate models, one for each agent. However, engineers are then faced with the burden of synchronizing and maintaining consistency among several models. Any change in the design may potentially impact all of the models. Furthermore, many multi-agent systems define a fixed boundary between the agents. These fixed boundaries are difficult to modify because they are often built into the system architecture; if the boundary is not appropriate, modification is difficult. These systems sometimes even use different modeling languages for each agent, in some cases with different semantics. This increases the cost of changing and validating models, and decreases the reliability of the overall system.

In this paper, we focus on specifying the scope of *planning* agents in multi-agent systems working on the same planning problem using a single declarative model. Having agents use the same model eliminates data synchronization problems, increases reuse, localizes changes easing maintainability, and reduces development cost and increases reliability. We provide a crisp and flexible characterization of the scope of each of the agents. This framework enables multi-agent system designers to specify the boundaries among agents. Furthermore, this framework makes it possible to use different design tradeoffs within the confines of a single system.

The paper is organized as follows. We first formalize the planning task in terms of *extensions* of an initial partial plan, and *flaws* that make a partial plan unacceptable to an agent. We then formally define *Flaw Filters* (FFs), a mechanism identifying a subset of all possible flaws that define the *necessary* work required by an agent to complete its' task. FFs induce *Plan Identification Functions* (PIFs), a means for an agent

*QSS Group, Inc.

†Research Institute for Advanced Computer Science

to determine when it has finished its planning task. Within this general framework it is possible to specify agent design further. We describe an implementation of PIFs and FFs used in EUROPA, a constraint-based planning framework. EUROPA assumes that all agents also share a *plan database* representing the current plan. We show how EUROPA's FF mechanism is used by agents to drive planning. We use a running example and describe some modeling issues that arise. Finally, we conclude and describe some future work.

The Planning Problem

In this section we describe a planning problem in a general way as required to define our framework. Many declarative planning domain languages have been created, and most of them share a set of common traits; rather than describe a particular language and paradigm, we loosely define common traits of all such languages and identify the core components needed to motivate the rest of our framework.

A *planning problem* consists of a *model* M and a *partial plan* P. The model consists of two parts. The first part is the set of *states* that describe the elementary building blocks of any plan. We take a loose view of state, and use the term to refer to either actions or fluents. The second part is a set of *rules* that describe relationships that must hold in any legitimate plan. The plan consists of a set of states and a set of relationships declared to hold between the states. A plan Q is an *extension* of P if $P \subset Q$. This means every state of P is a state of Q.

If we now take a plan P and a model M, we can determine which rules in M *apply* to P: if rule r mentions a state in P, then it applies. Rules may apply multiple times, and we can enumerate the instances of rule applications. A rule instance can be *provably satisfied*, *provably unsatisfied* or *not provably satisfied*. The set of rule instances that are not provably satisfied are called *flaws*. We divide flaws into two sets: $F_c(Q, M)$ is the set of provably unsatisfied flaws of Q under M, and $F_u(Q, M)$ is the set of not provably satisfied flaws.

We can now classify extensions of Q in a variety of ways. If Q is an extension of P and has no flaws, then it is a *solution* to P. The process of planning is then to find a solution of P. If $F_c(Q, M) = \emptyset$ we refer to Q as a *sound extension* of P. If $F_c(Q, M) \neq \emptyset$ we refer to Q as an *inconsistent extension* of P. If $F_u(Q, M) = \emptyset$ we refer to Q as an *complete extension* of P. Thus, a solution Q is a sound and complete extension of P.

Planning can proceed in one of two ways. The extension Q can be transformed into an extension of Q by adding states, or it can be transformed into S such that S is an extension of P and Q is an extension of S by removing states that are in $P - Q$. No state of P may be removed during planning. This is an extremely general notion of planning. It does not proscribe the form of the rules, nor does it proscribe how the rules are used to drive planning. Various common planning paradigms can be described using this framework, as we show below.

STRIPS motivates two styles of planning: linear and partial-order causal link (POCL) planning. Linear plans assume that all actions are totally ordered. STRIPS actions and fluents correspond to states in our formalism. A STRIPS planning problem statement consists of a STRIPS model with the initial and goal states, and an empty plan. Flaws arise from the STRIPS rules describing preconditions and effects of actions; if an action's preconditions are not the effects of some previous action or the initial fluents, the preconditions lead to flaws. A solution is found when all fluents are supported.

A POCL approach to planning allows concurrent actions. Concurrency gives rise to the possibility of clobbering effects of other concurrent actions, thus giving rise to causal link threats. In POCL states are augmented with causal links, and flaws are augmented to include threats.

Constructive search methods for planning are limited to generating sound extensions of P. Note that our formalization does not specify the direction in which planning is done nor does it specify the specific search method used. Thus, it covers progression and regression planning as well as various non-chronological backtracking and constructive sampling approaches like LDS. Local search methods such as those used by ASPEN (FRCY97) are typically concerned with generating complete extensions of P, and must resolve flaws in $F_c(Q, M)$.

IxTeT (GL94; LG95) is a planner that can reason about domains with time and resources. A plan is mapped to a Constraint Satisfaction Problem (CSP) and part of the planning is performed by searching for solutions to the CSP. Variables in this CSP are divided into temporal variables representing the start or end of activities, and atemporal variables representing attributes of states or actions in the plan. In IxTeT, a flaw is either an unfounded expression, an inconsistent expression, or a conflicting resource allocation. Unfounded expressions include new tasks required by existing tasks in the plan. Inconsistent expressions include violated constraints discovered by reasoning about the CSP. In our formalism, unfounded expressions belong to the set $F_u(Q, M)$; inconsistent expressions belong to the set $F_c(Q, M)$; and conflicting resource allocations belong to the set $F_c(Q, M)$ or $F_u(Q, M)$ depending on whether they can be proven inconsistent or not.

Plan Identification Functions

Suppose we are designing a set of agents to solve a planning problem P with a shared model M. All agents are assumed to know M, and we also assume that only one agent at a time will work on Q, which is some extension of P. There might be many criteria that force agents to hand over control to another agent, but we will limit ourselves to describing ways for agents to detect *necessary* conditions for handing over control. If an agent has no more work to do on a plan Q, then it must hand over control.

We define a *Flaw Filter* (FF) as a mapping from flaws to Y, N. If a flaw maps to Y then the agent must continue planning to resolve the flaw, otherwise the flaw can safely be ignored by this agent. This motivates a *Plan Identification Function* (PIF). A PIF is a function that maps partial plans P to the values Y and N. A return value of Y indicates a plan has no more filtered flaws, and therefore is ready to be passed on to another agent . A return value of N indicates a plan still has filtered flaws that the agent needs to resolve. A plan P *satisfies* a PIF i if $i(P)$ =Y. A PIF $i(P)$ induced by a FF f is denoted $i_f(P)$.

We distinguish the *null* FF as returning Y for all flaws. This induces the *null* PIF i_0 that returns Y only if the set of flaws is empty. This is the strictest possible interpretation of acceptability, and is used implicitly in most approaches to planning.

Properties of PIFs

When designing multi-agent systems we would like each agent to perform only part of the work. Since the model is shared among the agents, we would like to partition the flaws among agents designed to solve specific parts of the problem. Ultimately, the goal is to design a set of agents whose PIFs satisfy i_0.

Suppose we have a set of FFs used by a set of agents. The *union* of two filters $f_1 \cup f_2$ is the set of flaws F for which $f_1(f)$=y or $f_2(f)$=y. We can now reason about the joint behavior of agents using the following result:

Theorem 1 *Let M be a model and let P be a partial plan. Suppose $\mathcal{F}$ is a set of flaw filters. Suppose a collection of agents always finds an extension Q such that for each $f \in \mathcal{F}$ $i_f(P)$ =Y. Then the collection of agents satisfies $\cup_{f \in \mathcal{F}} i_f$.*

We can also order FFs and their PIFs . A flaw filter $f_1 \subset f_2$ if every flaw filtered by f_1 is filtered by f_2. This means $i_{f_1}(P) \Rightarrow i_{f_2}(P)$. Since fewer plans are accepted by $i_{f_1}(P)$ than $i_{f_2}(P)$, this induces an ordering on PIFs . Such an ordering might be used to dictate the order in which agents work on a planning problem.

We can measure the degree of overlap of two FFs . The *intersection* of two filters $f_1 \cap f_2$ is the set of flaws F for which $f_1(f)$=y and $f_2(f)$=y. This is work that either agent can do; this can be a feature if one agent runs out of time, or a problem if agents are supposed to all do disparate work, or if an agent is responsible for handling flaws it is not intended to handle. The intersection can let us determine how many agents can work on a plan, as well as help us design handoff strategies. Suppose we have two agents using FFs f_1 and f_2. A partial plan Q *links* the agents if $i_{f_1}(Q)$ =y and $i_{f_2}(Q)$ =n. Intuitively, this means that one agent is done with a planning problem and another agent is ready to work on the problem. This means that there exists a plan Q such that $f_1(Q) \cap f_2(Q) = \emptyset$. Suppose that $\forall Q$ $f_1(Q) \cap f_2(Q) = \emptyset$. In this case, only one of the two agents will ever need to work on the plan; such a pair of FFs is called *complementary.*

We now define some common characteristics that designers of PIFs may wish to enforce. In what follows, let i be the PIF . We assume we are defining the properties of i with respect to some planning problem P on a model M.

The first characteristic we define is consistency.

Definition 1 *A PIF, i, enforces* consistency *if, for any extension Q, such that $i(Q)$ = y, P has at least one solution.*

Suppose that i filters states that define Simple Temporal Networks (DMP91) that are guaranteed to be free of negative cycles. Such problems are known to have at least one solution, and such a PIF enforces consistency.

Another useful characterization is based how much work needs to be done by an execution engine to find a consistent completion in different circumstances. This is a particularly interesting question if uncertainty during execution is taken into account. The simplest case to handle is where the execution agent can make arbitrary choices to complete the given partial plan:

Definition 2 *A PIF, i, enforces* solvability *if, for any partial plan Q, such that $i(Q)$ = y, all extensions of Q are complete and consistent. Alternatively, Q defines a family of solutions of P.*

Suppose that i filters states that define Boolean Conjunctive Normal Form (CNF) problems for which all clauses are known to be satisfied, even though some clauses may have unassigned variables. This PIF enforces solvability.

A generalization of solvability relaxes the requirement that we find Q such that all extensions are solutions; instead, we need only to find Q such that all extensions satisfy the PIF .

Definition 3 *A PIF, i, enforces* i-solvability *if, for any partial plan Q, such that $i(Q)$ = y, all extensions R of Q have the property $i(Q)$ = y.*

Requiring solvability is often unnecessarily expensive for the planner and needlessly restrictive for the execution agent. A more relaxed notion is that a partial plan requires only a bounded amount of time to solve:

Definition 4 *A PIF, i, enforces* $O(f(n))$ solvability *if, for any partial plan Q satisfying $i(Q)$ = y, then in time $O(f(|Q|))$ either a solution of Q can be found or it can be shown that no consistent completion of Q exists.*

Suppose that i filters out Simple Temporal Networks. A simple polynomial time procedure can detect negative cycles in such problems; if such a cycle exists, the problem has no solution, and if no such cycles exist, then the problem has a solution. This PIF enforces polynomial time solvability.

As noted above, our formalization also covers agents that are capable of generating inconsistent plans. For such agents, the PIF may return y for inconsistent plans. In that case, it is useful to characterize the amount of time required to repair the plan, in order to find a consistent completion of the original problem.

Definition 5 *A PIF, i, enforces $O(f(n))$ transformability if, for any partial plan Q satisfying $i(Q) = \mathtt{y}$, then in time $O(f(|Q|))$ a solution to P can be found, or it can be shown that no solution exists.*

Suppose that i filters scheduling problems where the objective is to find a set of activities that can be scheduled in the face of temporal and resource constraints. Such problems are $\mathcal{NP}$-complete, so i enforces exponential-time transformability. The plan Q may contain activities that cannot be scheduled in the face of the existing constraints.

The Remote Agent: A Case Study in Plan Identification

The Remote Agent (JMM+00; MNP+98) was the first AI-based system to control a spacecraft in a two-day experiment in May of 1999. The Remote Agent consisted of three components: a Planner, an Executive, and a Mode Identification and Reconfiguration system. The Planner and Executive made use of PIFs , and (JMM+00) marks the first appearance of the concept in the literature. In this section we describe the PIFs used by these components.

The domain description language of the Remote Agent is similar to that of IxTeT, in that it represents domains with time and resources, and plans are mapped to CSPs. As with IxTeT, variables are divided into temporal variables and atemporal variables. Flaws in the Remote Agent Planner include uninstantiated atemporal variable flaws, open disjunctions (a special class of uninstantiated atemporal variables), floating states, and unresolved rules. All of these flaws fall into our class $F_u(Q, M)$. The class of temporal variable flaws was filtered out, as were flaws that fell strictly outside a *planning horizon*. The Planner implicitly handled $F_c(Q, M)$ using a form of backtracking search.

The Remote Agent Planner used a PIF that returned three values: `Y`,`N`,`?`. The value `Y` corresponded to a plan with $F_u(Q, M) = \emptyset$ and $F_c(Q, M) = \emptyset$, the value `N` corresponded to a plan with $F_c(Q, M) \neq \emptyset$, and `?` was returned otherwise. The plans satisfying the PIF consisted of Simple Temporal Networks that were guaranteed to have a solution; thus, the PIF for the Planner enforced *solvability*. The Executive was solely responsible for handling uninstantiated temporal variable flaws within a more limited planning horizon, and thus its PIF was both *linked* and *complementary* to the PIF of the Planner.

An Implementation of Plan Identification

We now turn our attention to describing an example system that implements a general architcture for using PIFs. The system, called EUROPA (Extensible Universal Remote Operations Planning Architecture), is an instantiation of the the CAIP *Constraint-based Attribute and Interval Planning* framework (FJ03) (CAIP). In this section, we first give an overview of EUROPA, then describe how PIFs are implemented as flaw filters. Further details on CAIP implementation can be found in (FJ03); in this section, we focus on those aspects that are most relevant to PIFs.

CAIP Overview

Plans in CAIP consist of *attributes* that can take on a sequence of values over time. Each attribute represents one of a number of concurrent threads describing the state of the world. An attribute takes on only one of a finite set of possible values at any time. Each of these values is a temporal interval. In CAIP, intervals represent actions and states with temporal extent. An *interval* is simply a predicate holding over a period of time. The start and end of the interval and the parameters of the predicate are described by variables. More formally, an *interval* is a tuple, (p, X, s, e), where p is a predicate name, X is a vector of variables defining the arguments to the predicate, and s and e are temporal variables, defining the start and end of the interval. In CAIP variables represent all aspects of states and actions and constraints to enforce relations between those variables.

A *planning domain* is defined by the set of interval types, and a set of configuration rules. A *configuration rule* is a generalization of the notion of preconditions and effects. It consists of a head and a set of consequences. The head of a configuration rule is a pattern for an action or a state. Each of the consequences specifies a state or action, along with a set of constraints among the variables of the head and the consequent. A configuration rule is *applicable* if its head matches some action or state in a plan. To satisfy the rule, all the consequences must also be in the plan, satisfying the associated constraints. The configuration rules are very expressive. Instead of specifying only state values before and after an action, they can specify arbitrary temporal relations between actions and states that must hold in a valid plan. These rules can also express disjunctions over consequents.

We will illustrate the concepts using an example of a simple spacecraft that can slew (i.e, turn to different orientations), take pictures, and download pictures to Earth. In order to take a picture of one of a specified number of objects, the spacecraft must be pointing at the target object. This will generally require slewing from one pointing to another. To ensure that residual vibration from the slew does not interfere with the picture, we require the slew to complete 10 seconds before taking the picture. The spacecraft's onboard data buffer may not be able to hold all of the pictures requested. In order to download pictures to Earth, the spacecraft will have to slew to Earth at one of a fixed set of times. An initial state for this problem consists of a set of objects, a set of picture requests, the times at which downlinking images can occur, and some constraints among the picture requests.

```
object ={Earth, asteroid-1, star-2, ...}
Attitude:{pointAt(object), turnTo(object),
idle()}
Camera:{off(), ready(), takePic(object)}
Take-Picture(B) → met-by ready()
Take-Picture(B)      →      contained-by[10,0]
pointAt(B)
ready() → met-by off()
pointAt(B) → met-by turnTo(B)
turnTo(B) → met-by idle()
```

Figure 1: A simple model of the spacecraft domain. The model consists of two attributes, `Attitude` and `Camera`. The attribute declarations are accompanied by a list of interval values that the attributes can take on. The rules specify temporal relationships between intervals drawn from Allen's Algebra, as well as describing (part of) a finite state machine governing the legal sequences each attribute can take on.

Figure 1 shows a fragment of a CAIP description of the satellite domain. For example, we have a configuration rule with a head specifying a `takePicture(x)` interval, and a consequence specifying that if I is such an interval, the plan must also contain a `pointAt(y)` interval, J, such that $x = y$, the start of J is at least 10 seconds before the start of I, and the end of J is no earlier than the end of I.

Let Q be a partial plan and let M be the model. We divide the not provably satisfied rule flaws $F_u(Q, M)$ into three different categories for convenience:

- Unbound temporal variables $F_v(Q, M)$. These consist of either s or e from some interval.
- Unbound parameter variables $F_p(Q, M)$. These are all other interval variables.
- Interval consequences that have not been inserted into the plan, $F_t(Q, M)$.

This partition is reminiscent of the scheme used to divide variable flaws in both IxTeT and the Remote Agent Planner.

Implementing Plan Identification in EUROPA

EUROPA is an implementation of CAIP that has been developed for use in planning space missions, both for single and multi-agent planning. PIFs provide considerable flexibility in designing agents for EUROPA. Engineers can design different PIFs and analyze the resulting performance of the integrated multi-agent system, and choose the PIF that works best for the application in question.

EUROPA commits to multi-agent planning with a *shared plan database*. All agents cooperating on a planning problem modify the same data structure (but not necessarily at the same time). Figure 2 shows the overall architecture of EUROPA. The system is composed of the following modules: a *planner*, a *plan database*, and a *plan identification* module. The plan identification module encapsulates both the filter criteria and the plan identification function. Planning begins when the plan dabase is initialized with a partial plan and a domain model. During planning, a planner can query the plan database through the plan identification module for filtered flaws in the current partial plan. The planner resolves filtered flaws by performing updates on the plan database. The plan database uses a constraint network to manage the consistency of variables and constaints and uses a temporal network to maintain consistency between temporal variables and the temporal relationships imposed by the configuration rules. If no flaws remain and the plan is consistent, the planner concludes that a plan has been found.

Figure 2: EUROPA system architecture diagram

Figure 3: Class Diagram of the PlanId Framework

Framework Class Diagram

Figure 3 presents the implementation details of the *PlanId* module referenced in Figure 2. There are three main components to the PIF implementation in EUROPA: a `FlawQuery`, `FilterCriteria`, and a `FlawCache`. A planner queries for filtered flaws through the `FlawQuery`. This component provides all access to filtered flaws by establishing a `Connection` with the `FlawCache`. All flaw changes since the last query are pushed from the `FlawCache` to the `FlawQuery` who then applies the `FilterCriteria` to select filtered flaws. The `FilterCriteria` object is just a collection of `Conditions` that specify which flaws pass the criteria. A filtered `Flaw` in satisfies all `Conditions`. Each `FlawQuery` has exactly one `FilterCriteria` instance, provided to it during construction. The FlawCache is synchronized with every plan database update. Notifications of changes in the contents of the `FlawCache`, i.e. flaws inserted or removed, are pushed to each connection from the `FlawCache` as the latter is synchronized with the `PlanDatabase`.

This architecture provides a number of useful features. First, the FlawCache can support many connections at once, enabling it to provide flaws to many planners. Second, a wide variety of simple conditions are provided, enabling a very large number of different PIFs to be expressible. Third, it is very straightforward to develop additional conditions making the approach very extendible. Finally, emphasis on lazy evaluation and event-based synchronization leads to efficient implementation.

We will describe the FF and PIF implementation of EUROPA for the satellite scheduling mission. We will

Figure 4: A simple partial plan for the model described in Figure 1

describe PIFs for three agents: a Science Planner, a Deliberative Planner, and an Executive. The FF for the Deliberative Planner, f_p, filters all flaws in $F_v(Q,M)$, that is, all temporal variable flaws. The FF for the Science Planner, f_s, also filters all temporal variable flaws. In addition, it filters elements of $F_t(Q,M)$ such that the predicate p of the interval is `off`, `idle` or `ready`. Let $d(v)_lb$ be the lower bound on the domain of variable v. The FF for the Executive, f_e, filters all temporal variable flaws such that $v \in F_v(Q,M)$ and $d(v)_{lb} > c$, a value that indicates an upper bound on the Executive's planning horizon. It also filters all flaws in $F_t(Q,M)$ and $F_p(Q,M)$.

The difference between the PIFs resides in each of the `FilterCriteria`. Each agents' criteria will use different conditions. For instance, the Deliberative Planner will use a variable condition allowing only flaws that are not temporal variables to pass the condition. The Science Planner will have an additional condition allowing only flaws that are neither variables of the predicate `TakePicture` or intervals representing this predicate, to pass the condition. Finally, the Executive will have a horizon condition allowing only flaws that are temporal variables that fall within the horizon to pass the condition.

```
while(done==false)
  if (isConsistent())
    filteredFlaws=getFlawsFromQuery()
    if (filteredFlaws.isEmpty()==false)
      nextFlaw = choose(filteredFlaws)
      resolve(nextFlaw)
    else done=true
  else... // rest of the algorithm omitted
end while
```

Figure 5: Planning with Flaw Queries. Each agent's planner will use its private Flaw Query to ensure that it receives only those flaws that matter. The FlawCache disseminates only the relevant flaws to each planner.

we've omitted the temporal variable flaws, putting them in will bloat this, what should we do? To see how the flaw filtering works, we will demonstrate how the Science Planner's PIF filters flaws during planning. Suppose planning begins with the sample partial plan shown in Figure 4. There are five flaws: the variables A, B and C, the `turnTo`(C) interval and the `pointAt`$(D = d)$ interval; the FlawCache has these five flaws. Recall that the PIF filters out intervals with predicate `off,ready` as well as all temporal variables. Then the set of filtered flaws after the first step consists of the three variable flaws and the `pointAt`$(D = d)$ interval.

Step 1:
  FlawCache=$\{A, B, C,$`pointAt`$(D = d)$`,turnTo`$(C)\}$
  FilteredFlaws:$\{A, B, C,$`pointAt`$(D = d)\}$
  nextFlaw: `pointAt`$(D = d)$
Step 2:
  FlawCache=$\{A, B, C, E,$`turnTo`(C)`,turnTo`$(E)\}$
  FilteredFlaws:$\{A, B, C, E,$`turnTo`$(E = d)$`, turnTo`$(C)\}$
  nextFlaw: `turnTo`$(E = d)$
Step 3:
  FlawCache=$\{A, B, C, E,\}$`turnTo`(C)`,idle()`$\}$
  FilteredFlaws:$\{A, B, C, E,,$`turnTo`$(C)\}$
  nextFlaw: A

Figure 6: Evolution of the flaws for the Science Planner during planning, beginning with the partial plan in Figure 4.

The basic loop of a planner is similar to the fragment presented in Figure 5. At each step, the planner requests the filtered flaws. Once the flaws are retrieved, the planner selects a flaw, then resolves it. Flaw resolution will often force propagation of variable changes in the constraint network that result in updates to the FlawCache. Subsequent queries to the FlawQuery will return a new set of flaws that accounts for these updates and the flaws filtered by the PIF.

To see this process in action, let us consider a few steps of planning given the partial plan and PIF that we have described. This process is shown in Figure 6. Let us assume that *choose* selects flaws according to some arbitrary order. Also suppose that *resolve* is the strategy for resolving a flaw. After deciding that `pointAt`$(D = d)$ should be part of the plan, the corresponding configuration rule ensures the creation of a `turnTo`(E) interval. This flaw is handled next, resulting in the creation at step 3 of an `idle()` flaw. The FlawQuery, however, indicates that the set of filtered flaws at step 3 omits this flaw. This is because the PIF for the SciencePlanner includes the condition that intervals with predicate `idle` don't pass. At the next step, choose() returns flaw $A$and planning proceeds. By contrast, the PIF for the Deliberative Planner would return `idle` as one of the flaws at step 3.

EUROPA's PIF framework supports the following conditions, among others:

- Interval predicate filtering - filters all intervals of a particular predicate.
- Interval variable filtering - filters selected variable of all intervals with a particular predicate.
- Temporal filtering - filters intervals according to a variety of temporal specifications. One example is a filter for intervals guaranteed not to happen within a temporal extent (a horizon filter).

Complexity Analysis

In the simplest PIF implementation, one could omit the `FlawCache` and `Connection` infrastructure. Resolving a query would be accomplished by iterating over all intervals and variables in the plan database and for each, applying the filter to test for inclusion or exclusion. This would result in a worst-case time-complexity given by $(N_v + N_i) * N_c * C_c$ where N_v is the number of variables, N_i is the number of intervals, N_c is the number of conditions in the filter, and C_c is the average cost of evaluating a condition. [1]

Since the points of greatest cost are in the evaluation of conditions, we seek to reduce the execution of condition tests. This is accomplished in a numnber of ways: 1. The last set of filtered flaws are cached in each `FlawQuery`; 2. The current set of flaws in the plan database are cached in the `FlawCache`; 3. Each cache is maintained through notifications of changes; 4. Conditions may be ordered to fail fast, based on the characteristcs of each problem; 5. The FlawQuery is updated only when the planner requests the latest set of flaws. Thus, the queries are only run on the set of flaws that were added since the last query.

The resulting worst-case cost of a query is approximated by: $N_+ * N_c * C_c$ where N_+ is the number of flaws inserted into the flaw cache since the last query.[2] The approximation omits the cost of caching events during synchronization of the FlawCache and the PlanDatabase. This is reasonable since the costs of caching are much less than the cost of evaluating the conditions over all insertions. Notice that we do not need to worry about flaws that are removed from the cache, since they aren't returned to the planner in any case.

Related Work

Many integrated planning and plan execution frameworks define a fixed boundary between their components. These systems also use different modeling languages, in some cases with different semantics, and thus have potential problems with model synchronization. Finally, these systems do not have a crisp declarative characterization of the boundary between the components. This means that redistribution of the duties of a component is either impossible or requires expensive model revision. Examples of integrated planning and plan execution systems in this category are O-Plan (TDK94), Cypress (WMLW95) 3T (BFG$^+$97), Propice-Plan (DI99) and the Remote Agent (MNP$^+$98; JMM$^+$00).

IDEA (MDF$^+$02; DLM03) is an agent architecture designed to overcome shortcomings in the Remote Agent approach to agent modeling. IDEA provides a simple virtual machine that supports plan execution, consisting of a model, plan database, plan runner, and reactive planner. The job of the reactive planner component of an IDEA agent is to ensure that a "locally executable" plan is returned to the plan runner. Thus, a crucial task is to define the scope of the reactive planner's job. The PIF is a natural way to focus on those parts of the model that must be addressed by each IDEA agent. IDEA also supports the notion of multiple planners operating on the same plan database, and thus the same model. In particular, the plan runner can be thought of as an Executive, while a reactive planner can perform almost any task. PIFs are a natural way to define the scope of these various planners in order to ensure that planners do not step on each others' toes.

[1] In practice only some of the conditions will be executed since we discard the flaw after the first condition fails.

[2] $N_+ << (N_i + N_v)$ since there are relatively few flaw insertions resulting from each planner commitment.

Discussion and Future Work

This framework presents a means by which a single model can be used and shared among all of the agents. It enables an adjustable definition of the amount of work each agent will perform without revising the declarative model of the planning domain. Having an easy and low-overhead way to adjust the scope of work of each agent simplifies the design of multi-agent systems and reduces development and validation cost.

The notion of PIFs is lacking in some formality. EUROPA's implementation of PIFs , for example, depends on details of the plan structure to distinguish between different categories of flaws. For example, flaws are distinguished by the predicate of intervals, and by variable type (e.g. temporal variables). In part, this is a manifestation of the fact that we have skimmed over the relationship between flaws and rules. This crucially depends on the exact details of the plan structure. We have purposefully given this issue a brief treatment, but more formality is needed.

The notion of PIFs leaves aside any consideration of how to coordinate among the agents responsible for finding plans. In particular, a more careful examination of the handoff strategies agents can employ given a set of PIFs is worthwhile.

References

R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experienences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Ingelligence*, 9(2), 1997.

O. Despouys and F. Ingrand. Propice-plan: Towards a unified framework for planning and execution. In *Proceedings of the 5th European Conference on Planning*, 1999.

M. Dias, S. Lemai, and N. Muscettola. A real-time rover executive based on model-based reactive planning. In *Proceedings of the International Conference on Robotics and Automation*, 2003.

R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–94, 1991.

J. Frank and A. Jónsson. Constraint based attribute and interval planning. *Journal of Constraints*, To Appear, 2003.

A. Fukunaga, G. Rabideau, S. Chien, and D. Yan. Toward an application framework for automated planning and scheduling. In *Proceedings of the* 15th *International Joint Conference on Artificial Intelligence*, 1997.

M. Ghallab and H. Laurelle. Representation and control in ixtet, a temporal planner. In *Proceedings of the 4th International Conference on AI Planning and Scheduling*, pages 61–677, 1994.

A. Jónsson, P. Morris, N. Muscettola, K. Rajan, and B. Smith. Planning in interplanetary space: Theory and practice. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, 2000.

P. Laborie and M. Ghallab. Planning with sharable resource constraints. In *Proceedings of the* 14th *International Joint Conference on Artificial Intelligence*, 1995.

N. Muscettola, G. Dorais, C. Fry, R. Levinson, and C. Plaunt. Idea: Planning at the core of autonomous reactive agents. In *Proceedings of the 3d International NASA Workshop Planning and Scheduling for Space*, 2002.

N. Muscettola and P. Morris. Execution of temporal plans with uncertainty. In *Proceedings of the* 17th *National Conference on Artificial Intelligence*, 2001.

P. Morris, N. Muscettola, and I. Tsamardinos. Reformulating temporal plans for efficient execution. In *Proceedings of the* 15th *National Conference on Artificial Intelligence*, 1998.

N. Muscettola, P. Morris, and T. Vidal. Dynamic control of plans with temporal uncertainty. In *Proceedings of the* 17th *International Joint Conference on Artificial Intelligence*, 2001.

N. Muscettola, P. Nayak, B. Pell, , and B. Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1-2), 1998.

A. Tate, B. Drabble, and R. Kirby. O-plan2: An open architecture for command, planning and control. *Intelligent Scheduling*, 1994.

D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical Artificial Ingelligence*, 7(1), 1995.